Everton Rocha Pereira da Silva

Desenvolvimento Guiado por Testes

São Paulo

2012

Everton Rocha Pereira da Silva

Desenvolvimento Guiado por Testes

Monografia apresentada á Faculdade de Tecnologia de São Paulo para obtenção do titulo de Tecnólogo em Processamento de Dados

Orientador

Cecília Keiko Adati Tomomitsu

São Paulo

2012

DEDICATÓRIA

Única e exclusivamente a minha mãe Dilma Rocha da Silva, sem a qual não teria chegado até aqui.

AGRADECIMENTO

Aos meus amigos desenvolvedores, a comunidade DevBrasil e todos que colaboraram e me motivaram para conclusão desse trabalho.

RESUMO

Entregar software com qualidade é um dos maiores desafios do desenvolvimento de sistemas atualmente. Todos os anos bilhões de dólares são desperdiçados por causa da baixa qualidade dos sistemas desenvolvidos, por causa das falhas de software.

Para enfrentar esses problemas, este trabalho propõe a utilização da pratica Desenvolvimento guiado por testes, pratica que compõe o núcleo da metodologia ágil Extreme Programming, para aumentar a qualidade de software diminuindo o índice de falhas.

Neste trabalho é apresentada a teoria do Desenvolvimento guiado por testes e também é conduzido um estudo de caso para evidenciar os resultados obtidos através dessa pratica

Palavras-chave: TDD, Desenvolvimento guiado por testes, Qualidade de software, Testes unitários.

ABSTRACT

Delivering quality software is one of the biggest challenges in developing systems today. Every year billions of dollars are wasted because of poor quality of the systems developed, because of software failures.

To address these problems, this paper proposes the use of test-driven development practices, practices that make up the core of agile Extreme Programming, to increase software quality by reducing the failure rate.

This work presents the theory of development driven by testing and also conducted a case study to highlight the results obtained through this practice

Keywords: TDD, test-driven development, software quality, testing unit.

Lista de figuras

Figura 1 - Testes dentro da XP ..... **Erro! Indicador não definido.**

Figura 2 - Ciclo TDD ..... **Erro! Indicador não definido.**

Figura 3 - Objeto Simulado ..... **Erro! Indicador não definido.**

Figura 4 - Codigo Primo 1 ..... **Erro! Indicador não definido.**

Figura 5 - Codigo Primo 2 ..... **Erro! Indicador não definido.**

Figura 6 - Codigo Primo 3 ..... **Erro! Indicador não definido.**

Figura 7 - Código Primo 4 ..... **Erro! Indicador não definido.**

Figura 8 - Código Primo 5 ..... **Erro! Indicador não definido.**

Figura 9 - Código Primo 6 ..... **Erro! Indicador não definido.**

Figura 10 - Codigo Primo 7 ..... **Erro! Indicador não definido.**

Figura 11 - Codigo Primo 8 ..... **Erro! Indicador não definido.**

Lista de tabelas

Tabela 1 – Tabela de Índice de Manutenibilidade..... **Erro! Indicador não definido.**

Tabela 2 – Tabela de Complexidade Ciclomática.....41

Tabela 3 – Tabela de Acoplamento.....41

Sumario

Lista de figuras 7

Lista de tabelas 8

Sumario 9

1 INTRODUÇÃO 13

1.1 Contextualização 13

- Defeito 13

1.2 Formulação do problema 16

1.3 Objetivo Geral 16

1.4 Objetivos específicos 16

1.5 Contribuição 16

1.6 Metodologia 16

1.7 Delimitação do tema 16

1.8 Estrutura do trabalho 17

2 METODOLOGIAS ÁGEIS 12

2.1 Definição 12

2.2 Manifesto Ágil 12

2.3 Extreme Programming 13

2.4 Valores 13

2.4.1 Feedback 13

2.4.2 Comunicação 20

2.4.3 Simplicidade 20

2.4.4 Coragem 21

2.5 Praticas 21
2.5.1 Cliente Presente 21
2.5.2 Jogo do Planejamento 22
2.5.3 Pequenos releases 23
2.5.4 Testes de cliente 23
2.5.5 Projetos Simples 24
2.5.6 Programação em par 25
2.5.7 Desenvolvimento guiado por testes 25
2.5.8 Projeto de melhoria 25
2.5.9 Integração Continua 25
2.5.10 Propriedade de código coletiva 26
2.5.11 Padrão de codificação 26
2.5.12 Ritmo sustentável 26

3 TESTE DE SOFTWARE 27

3.1 Métodos de Teste 27
3.1.1 Teste Caixa-preta 27
3.1.2 Testes Caixa-branca 27
3.2 Principais Tipos de Teste de Software 29
3.2.1 Testes Unitários 29
3.2.2 Testes de Aceitação 29
3.2.3 Testes de Integração 29
3.2.4 Testes de Regressão 30

4 DESENVOLVIMENTO GUIADO POR TESTES 29

4.1 Teoria 32
4.2 Ciclo TDD 32
4.3 Lista de Testes 33
4.4 Assertivas 33
4.5 Objetos Simulados 34
4.6 Stubs 35

5 Frameworks XUnit 36

5.1 SUnit 36
5.3 Exemplo do Cliclo TDD com NUnit 37

6 Estudo de Caso 43

6.1 Hospital S* 43
6.2 Empresa F* 43
6.3 O projeto sem papel 43
6.4 Ambiente de utilização do sistema 44
6.5 Ambiente de Desenvolvimento 44
6.6 Métricas de Software 44
6.6.1 Indice de Manutenabilidade 44
6.6.2 Complexidade Ciclomática 45
6.6.3 Acoplamento de classes 45
6.6.4 Cobertura de testes 45
6.7 Valores coletados 46
6.7.1 Índice de Manutenibilidade 46
6.7.2 Complexidade Ciclomática 46
6.7.3 Acoplamento 46
6.8 Analise dos Resultados 47

7 Conclusão 49

1 INTRODUÇÃO

Neste tópico serão abordados os problemas do desenvolvimento de software, mais especificamente os problemas derivados das falhas de software demonstrando dados estatísticos que comprovam grandes prejuízos financeiros e também será apresentada uma alternativa para desenvolvedores aumentarem a qualidade de software.

1.1 Contextualização

Desenvolver softwares de qualidade tem sido um desafio desde o inicio da era da computação.

(SOARES, Michel Dos Santos. **Metodologias Ágeis Extreme Programming e Scrum para o Desenvolvimento de Software)**[1]

Em 2009 o conhecido relatório CHAOS REPORT (Standish Group) mostra essa realidade em números:

- 24% dos projetos fracassam
- 44% dos projetos são entregues com sucesso parcial
- E apenas 32% dos projetos obtêm sucesso.

Por esses motivo, diversas tentativas para melhorar a qualidade do software tem sido feitas, como a criação da disciplina Engenharia de Software (VASCONCELOS, Alexandre Marcos Lins de et al. **Introdução á engenharia de software e á qualidade de software.**)[2].

"Resumidamente, "qualidade" é conformidade com requisitos, e estes devem estar definidos para permitir que sejam gerenciados com o uso de medidas, de forma a reduzir o retrabalho e aumentar a produtividade (GUERRA, Ana Cervigni; COLOMBO, Regina Maria Thienne. **Tecnologia da Informação: qualidade de produto de software.)** [3]

[1] **.** Disponível em: < revistas.facecla.com.br/index.php/reinfo/article/download/146/38>. Acesso em: 24 jun. 2012.

[2] Disponível em: <http://www.cin.ufpe.br/~if720/downloads/Mod.01.MPS_Engenharia&QualidadeSoftware_V.28.09.06.pdf>. Acesso em: 24 jun. 2012.

[3] Disponível em: < **http://hdl.handle.net/10691/151**>. Acesso em: 19 jun. 2012.).

Os requisitos são necessidades explicitas do cliente e para atendê-las e conseguir a satisfação do cliente é imprescindível o desenvolvimento de um produto com a ausência de defeitos, erros ou falhas (GUERRA, Ana Cervigni; COLOMBO, Regina Maria Thienne. **Tecnologia da Informação: qualidade de produto de software.)** [4]**.**

A norma IEEE-610 define defeitos, erros e falhas da seguinte forma:

- **Defeito**

  Ato inconsistente cometido por um indivíduo ao tentar entender uma determinada informação, resolver um problema ou utilizar um método ou uma ferramenta.
  Pode ocasionar a manifestação de erros em um produto.
  Instrução ou comando incorreto (hardware/software fault).
  Causa raiz é sempre o defeito (a falta).

- **Erro**

  Manifestação concreta de um defeito num artefato de software.
  Qualquer estado intermediário incorreto ou resultado inesperado na execução de um programa constitui um erro.
  Diferença entre o valor obtido e o valor esperado.
  Construção de um software de forma diferente ao que foi especificado (universo de informação).

- **Falha**

  Comportamento operacional do software diferente do esperado pelo usuário.
  Diferença indesejável entre o observado e o esperado, é um evento.
  Uma falha pode ter sido causada por diversos erros e alguns erros podem nunca causar uma falha.
  Afetam diretamente o usuário final da aplicação (universo do usuário).
  Pode inviabilizar a utilização de um software.
  Estado intermediário de instabilidade podendo resultar em uma falha.

[4] Disponível em: < **http://hdl.handle.net/10691/151**>. Acesso em: 19 jun. 2012.).

A imagem abaixo ilustra bem o ambiente em que estão defeitos, erros e falhas.

Figura 1 - Defeito, erro e falha.

Fonte: http://fabiano-falcao.blogspot.com.br/2011/11/diferenca-entre-erro-defeito-e-falha-no.html

Falhas de software é um grande problema e anualmente causam grandes perdas financeiras, uma pesquisa realizada pelo departamento de Comercio dos EUA (DEPARTMENT OF COMMERCE (Usa). **Software Errors Cost U.S. Economy $59.5 Billion Annually.**)[5] revelou que falhas de software custavam 59.5 bilhões de dólares a economia americana anualmente e mais que 22,2 bilhões poderiam ser economizados com uma infraestrutura melhor para identificação mais cedo dos erros.

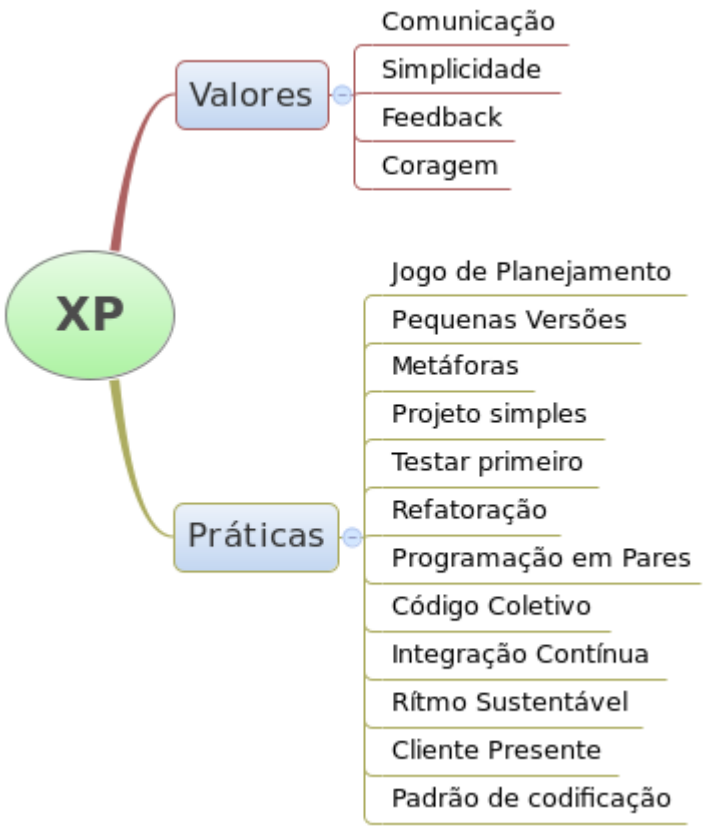
Este trabalho se propõe a apresentar o Desenvolvimento guiado por testes que é uma pratica de desenvolvimento que pode contribuir para aumentar a qualidade do software, identificando erros mais cedo e assim contribuir para diminuir custos.

O Desenvolvimento guiado por testes, Test-Driven Development em inglês ou simplesmente TDD como é comumente conhecido é uma pratica que se popularizou com metodologias ágeis, especialmente com Extreme Programming (XP) que considera TDD uma de suas praticas mais importantes (SANTOS, 2010, p. 4).

[5] Disponível em: <http://web.archive.org/web/20090610052743/http://www.nist.gov/public_affairs/releases/n02-10.htm>. Acesso em: 02 jun. 2012

A metodologia XP é fundamentada em valores e praticas (PIRES, Cassio. **Extreme Programming: um novo conceito em Metodologia de Desenvolvimento.**)[6], a imagem abaixo mostra como se estrutura a programação extrema.

Figura 2 - Programação Extrema



Fonte: Santos, 2010

Praticas do XP como integração continua, pequenos releases, código coletivo, refatoração poderiam causar uma serie de problemas, poderiam gerar defeitos no software, mas não é isso que ocorre na pratica, isso deve-se a obseção que XP tem por testes (JEFFRIES, Let Ron. **What is Extreme Programming?**)[7], nesse ponto podemos notar o porque TDD é considerado uma das praticas mais importantes da XP.

[6] Disponível em: <http://www.gojava.org/files/artigos/ExtremePrograming.pdf>. Acesso em: 02 jun. 2012.

[7] Disponível em: <http://xprogramming.com/what-is-extreme-programming/#planning>. Acesso em: 02 jun. 2012.

1.2 Formulação do problema

O desenvolvimento guiado por teses é capaz de aumentar a qualidade do software diminuindo o numero de falhas de software?

1.3 Objetivo Geral

Avaliar a qualidade do software desenvolvido com TDD

1.4 Objetivos específicos

Apresentar Principais tipos de testes.

Apresentar principais ferramentas xUnit.

Levantamento e coleta de dados.

Análise dos dados coletados.

Apresentar indicadores específicos para evidenciar que a abordagem TDD aumenta a qualidade do software?

1.5 Contribuição

O estudo do desenvolvimento guiado por teste deverá apresentar uma abordagem para desenvolvimento de software e apresentar uma alternativa para aumentar a qualidade do software desenvolvido.

1.6 Metodologia

Para atingirmos nosso objetivo de avaliar a pratica TDD, necessitamos conhecer melhor os problemas encontrados no desenvolvimento de software, e a pesquisa bibliográfica fornecerá a base teórica para a aplicação da técnica TDD, a maneira correta de sua aplicação. Após estarmos familiarizados com o problema e conhecermos toda a teoria do desenvolvimento guiado por testes poderemos enfim testar essa técnica através de um estudo de caso e recolher informações e avaliar os resultados dessa pratica em determinado cenário (GIL, 1991).

1.7 Delimitação do tema

Análise dos resultados obtidos através da aplicação dos conceitos do Desenvolvimento Guiado por Testes.

Muitos fatores podem causar falhas de software, nesse trabalho se restringirá a analisar apenas erros decorrentes de codificação, ou seja, da programação. Erros de analise de requisito, regras de negocio entre outros não serão apreciadas nesse trabalho.

1.8 Estrutura do trabalho

No segundo capítulo será apresentada brevemente a programação extrema onde poderemos conhecer um pouco do ambiente em que o TDD popularizou. Em seguida, no terceiro capitulo, conheceremos os principais tipos de teste de software que utilizaremos para desenvolver com TDD e para avaliar o software produzido.

O quarto capítulo irá apresentar o desenvolvimento guiado por testes, o ciclo de desenvolvimento e sua teoria. No quinto capitulo será apresentado as ferramentas XUnit e um exemplo de código com TDD.

O sexto capítulo ira apresentar um estudo de caso onde iremos procurar identificar alterações na qualidade do software com a utilização da pratica TDD.

O sétimo e ultimo capitulo apresentara a conclusões finais

2 METODOLOGIAS ÁGEIS

Em 2001 Kent Beck e outros 16 notáveis profissionais de TI reuniram-se e assinaram o que veio a ser conhecido como o manifesto ágil. (MANIFESTO AGIL. **Manifesto para o desenvolvimento ágil de software.**)[8]

Muitas das ideias do manifesto ágil já existiam há algum tempo, mas somente na década de 1990 que essas ideias vieram a se cristalizar em um movimento de desenvolvimento de software (Pressman, 2006, p 58).

2.1 Definição

As metodologias de desenvolvimento chamadas metodologias ágeis surgiram nos anos noventa com a proposta de conseguir melhores resultados que as metodologias chamadas tradicionais, livrando-se de alguns conceitos para conseguir atender as mudanças que acontecem durante o processo de desenvolvimento de software.

> A engenharia de software ágil combina uma filosofia e um conjunto de diretrizes de desenvolvimento. A filosofia encoraja a satisfação do cliente e a entrega incremental do software logo de inicio; equipes de projetos pequenas, altamente motivadas; métodos informais; produtos de engenharia mínimos e simplicidade global do desenvolvimento (PRESSMAN, 2006, p. 58).

2.2 Manifesto Ágil

Em um grupo de excepcionais profissionais de TI reuniram-se em uma estação de esqui nos Estados Unidos para discutir melhores praticas para o desenvolvimento de sistemas, embora cada um possuísse suas próprias técnicas para desenvolvimento, eles conseguiram definir princípios gerais para o desenvolvimento (IT, Improve. **Manifesto Ágil**)[9]. Eles declararam:

Indivíduos e interações entre eles mais que processos e ferramentas;

Software em funcionamento mais que documentação abrangente;

Colaboração com o cliente mais que negociação de contratos;

Responder a mudanças mais que seguir um plano.

[8] Disponível em: <http://agilemanifesto.org/iso/ptbr/>. Acesso em: 24 jun. 2012.
[9] Disponível em: <http://improveit.com.br/xp/manifesto_agil>. Acesso em: 19 jun. 2012.

Isto é, ainda que haja valor nos itens à direita, valorizamos mais os itens à esquerda. (IT, Improve. **Manifesto Ágil**).

Hoje temos varias metodologias ágeis, ou seja, metodologias que seguem os princípios do manifesto ágil e entre elas, uma ganha destaque pelo numero de adeptos e projetos á Extreme Programming (SOARES, Michel Dos Santos. **Metodologias Ágeis Extreme Programming e Scrum para o Desenvolvimento de Software.**).[10]

2.3 Extreme Programming

Extreme Programming é uma das metodologias ágeis mais conhecidas e utilizadas, segundo Kent Beck a programação extrema foi concebida para abordar necessidades especificas de equipes pequenas com projetos com requisitos vagos e instáveis (Beck, 2004).

A XP vem fazendo sucesso em diversos países, ajudando a desenvolver sistemas com maior qualidade e gastando menos tempo. Esse sucesso é conseguido seguindo um pequeno grupo de valores e praticas (IT, Improve. **Extreme Programming**)[11].

2.4 Valores

Segundo Pires (PIRES, Cassio. **Extreme Programming: um novo conceito em Metodologia de Desenvolvimento**)[12] os valores definem como as equipes devem proceder durante o desenvolvimento e frisa a importância de observa-los para conseguir o melhor resultado dessa metodologia, logo abaixo podemos conhecer os valores da XP.

2.4.1 Feedback

Nas metodologias tradicionais é muito comum ao final de um projeto ouvir do cliente a conhecida frase “Não era bem isso que eu queria”, isso ocorre porque com o cliente distante o feedback somente é dado no final do projeto ou de cada fase, dessa forma é mais difícil corrigir eventuais problemas. XP propõe uma parceria com o cliente durante o desenvolvimento, exige a participação do cliente no projeto e assim seu feedback imediato para homologar as funcionalidades.

---

[10] **.** Disponível em: < revistas.facecla.com.br/index.php/reinfo/article/download/146/38>. Acesso em: 24 jun. 2012.

[11] Disponível em: <http://improveit.com.br/xp>. Acesso em: 19 jun. 2012.

[12] Disponível em: <http://www.gojava.org/files/artigos/ExtremePrograming.pdf>. Acesso em: 02 jun. 2012.

2.4.2 Comunicação

No desenvolvimento de software, um dos pontos críticos é a comunicação porque desenvolver software não tem a ver com linhas de código e sim com lidar com pessoas.

> Projetos de software normalmente envolvem a presença de pelo menos duas pessoas, um usuário e um desenvolvedor, o que causa a necessidade de comunicação entre elas. No mínimo, cabe ao usuário comunicar o que necessita que seja produzido e ao desenvolvedor comunicar as considerações técnicas que afetam a solução e a velocidade de implementação da mesma. (TELES, 2005, p.59.).

Em projetos grades em que muitas pessoas e muitas áreas estão envolvidas gerenciar a comunicação se torna um dos grandes desafios.

Na busca de uma comunicação eficiente XP busca sempre uma comunicação direta entre cliente e desenvolvedores, exigindo que o cliente esteja presente ou alguém que o represente dessa forma existe um ganho em produtividade, pois os desenvolvedores tem acesso direto às informações que precisam.

Quando se utiliza a metodologia XP é imprescindível fazer com que o cliente compre a ideia de participar do projeto diretamente, o que às vezes é complicado, pois o cliente geralmente não esta acostumado a participar ativamente do desenvolvimento do sistema ou disponibilizar alguém para representa-lo frente à equipe de desenvolvimento.

2.4.3 Simplicidade

Equipes que utilizam XP buscam sempre a simplicidade no desenvolvimento focando-se no que realmente importa, evitando assim que tempo e esforço seja gasto inutilmente tentando evitar possíveis problemas ou alterações futuras.

“Equipes que utilizam XP buscam sempre a simplicidade no desenvolvimento focando-se no que realmente importa, evitando assim que tempo e esforço seja gasto inutilmente tentando evitar possíveis problemas ou alterações futuras.” (TELES, 2005).

2.4.4 Coragem

No desenvolvimento de software, tanto desenvolvedores quanto o cliente tem suas preocupações e essas preocupações são naturais dentro de um desenvolvimento de um sistema. Equipes XP lidam com medo com a coragem adquirida pela confiança nas praticas do XP.

"É necessário ter coragem para lidar com esse risco, o que em XP se traduz em confiança nos seus mecanismos de proteção." (PIRES, Cassio. **Extreme Programming: um novo conceito em Metodologia de Desenvolvimento**)[13].

"Ter coragem em XP significa ter confiança nos mecanismos de segurança utilizados para proteger o projeto." (TELES, 2005, p. 67).

Metodologias ágeis rompem com uma serie de comportamentos, princípios e regras do desenvolvimento em cascata ou estruturado por esse motivo é necessária coragem para abandonar essas praticas e adotar novas praticas, praticas ágeis.

2.5 Praticas

Praticas em XP representam aquilo que as equipes que utilizam extreme programming fazem diariamente durante o processo de desenvolvimento e podemos selecionar quais delas utilizar em uma determinado contexto, "Se a situação muda, você seleciona diferentes praticas para abordar essas condições"(IT, Improve. **Praticas.**).[14]

2.5.1 Cliente Presente

Como já foi visto anteriormente um dos valore do XP é a comunicação e não é possível uma comunicação efetiva com alguém ausente, email e vídeos conferências não é tão produtivo quanto uma conversa cara a cara.

"O XP, ao contrário, propõe que o cliente esteja presente no dia-a-dia do projeto. Sua presença torna o processo de desenvolvimento muito mais simples, conduzindo o desenvolvimento com pequenos ajustes ao longo do projeto. " (PIRES, Cassio. **Extreme Programming: um novo conceito em Metodologia de Desenvolvimento**.)[15]

---

[13] . Disponível em: <http://www.gojava.org/files/artigos/ExtremePrograming.pdf>. Acesso em: 02 jun. 2012
[14] Disponível em: <http://improveit.com.br/xp/praticas>. Acesso em: 19 jun. 2012.
[15] **.** Disponível em: <http://www.gojava.org/files/artigos/ExtremePrograming.pdf>. Acesso em: 02 jun. 2012

Com o cliente próximo ao desenvolvimento o feedback é imediato e ganhamos a possibilidade de corrigir erros antes da entrega final.

“Em termos práticos, isso significa colocar o cliente fisicamente próximo aos desenvolvedores ou mover os desenvolvedores para próximo do cliente.” (TELES, 2005, p. 70).

2.5.2 Jogo do Planejamento

O planejamento em XP é utilizado para manter o foco no que é mais importante para o projeto, o cliente escolhe as estórias que gostaria que fossem implementadas e a equipe com iterações de até duas semanas, a equipe entrega uma funcionalidade solicitada pelo cliente.

“Estas etapas de planejamento são muito simples, mas eles fornecem informações muito boa e excelente controle de direção nas mãos do cliente. A cada duas semanas, a quantidade de progresso é totalmente visível.”(JEFFRIES, Let Ron. **What is Extreme Programming?)**[16]

Esse processo gera uma confiança do cliente para com o projeto, pois ele pode ver o trabalho sendo feito, pode ver o progresso.

“Em XP o cliente é o responsável pelas decisões de negócio enquanto que os desenvolvedores, pelas decisões técnicas. As funcionalidades do sistema são descritas pelas

estórias que o cliente escreve à mão em pequenos cartões.” (PIRES, Cassio. **Extreme Programming: um novo conceito em Metodologia de Desenvolvimento.)**[17]

O cliente escreve estórias em cartões e essas estórias serão traduzidas em funcionalidades e tarefas. Quando o cliente escreve essas estórias ele se sente responsável por ela e por isso costuma avaliar muito bem se o que esta pedindo é realmente importante.

> Muitas vezes as estórias produzem sistemas simples que podem ser implementadas rapidamente. No entanto há estórias que podem consumir muito esforço de desenvolvimento, demorando dias ou até semanas. Para esses casos as estórias são reescritas em tarefas, as quais são divididas entre os desenvolvedores. (PIRES, Cassio. Extreme Programming: um novo conceito em Metodologia de Desenvolvimento) [18].

[16] Disponível em: <http://xprogramming.com/what-is-extreme-programming/#planning>. Acesso em: 02 jun. 2012).
[17] Disponível em: <http://www.gojava.org/files/artigos/ExtremePrograming.pdf>. Acesso em: 02 jun. 2012)
[18] Disponível em: <http://www.gojava.org/files/artigos/ExtremePrograming.pdf>. Acesso em: 02 jun. 2012.

2.5.3 Pequenos releases

XP trabalha com o conceito de pequenos releases. Equipes XP fazem entregas frequentes de novas funcionalidades agregando valor ao negocio, dessa forma o cliente pode testar e decidir se por colocar em produção. (EXTREME PROGRAMING.ORG (Usa). **Make frequent small releases**)[19].

Desenvolver incrementalmente teoricamente deveria causar todos tipos de problemas na hora implantar uma nova funcionalidade no sistema, mas nao é isso que acontece na pratica com XP.

"[...] estes lançamentos freqüentes são mantidos confiáviaveis pela obsessão do XP com os testes" (JEFFRIES, Let Ron. **What is Extreme Programming?**).[20]

Nesse ponto o desenvolvimento guiado por teste merece destaque, os teste unitarios garantem que as funcionalidades anteriormente implementadas continuarao funcionando apos a integração com os novos modulos.

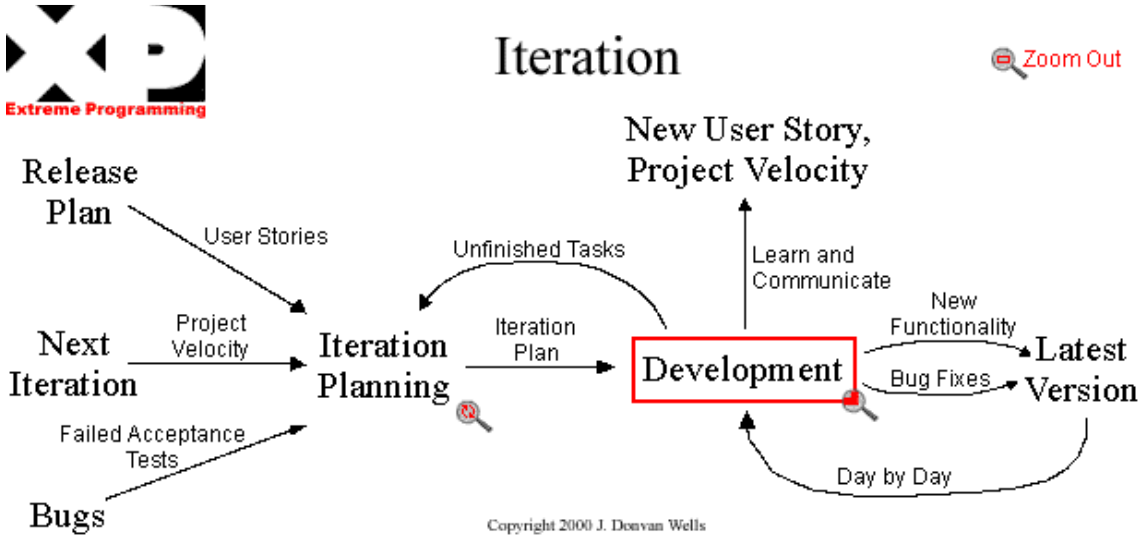
2.5.4 Testes de cliente

Para por a prova para o cliente que uma nova funcionalidade foi implementada são criados testes de aceitação, testes automatizados, para que o cliente teste, validando ou não se a estória foi implementada corretamente. O cliente é responsável por decidir se o teste falhou ou não.

Esses testes também são mantidos funcionando após o teste do cliente assim garantindo uma evolução constante do sistema.

[19] . Disponível em: http://www.extremeprogramming.org/rules/releaseoften.html>. Acesso em: 02 jun. 2012.
[20] Disponível em: <http://xprogramming.com/what-is-extreme-programming/#planning>. Acesso em: 02 jun. 2012

Figura 3 - Testes dentro da XP



Fonte: http://www.extremeprogramming.org/

2.5.5 Projetos Simples

O relatório do Standish Group mostra que 19% das funcionalidades raramente são usadas e 45% nunca são utilizadas, ou seja, 64% das funcionalidades são inúteis.

Manter um projeto simples é uma tarefa árdua, é comum no inicio do projeto o cliente não ter certeza do que realmente precisa por esse motivo acaba solicitando mais do que realmente precisa. Entre os desenvolvedores também surgem problemas, constantemente ficamos tentados a implementar essa ou aquela funcionalidade por acharmos que o cliente ira solicitar em algum momento ou tentarmos generalizar ao Maximo o sistema para possíveis alterações futuras.

“Essas preocupações são justificáveis, porem as abordagens utilizadas apresentam problemas”. (TELES, 2005, p. 64).

A maneira correta de enfrentar esses problemas em XP é manter o foco na implementação da estória escolhida para a iteração, não se preocupar com o que poderá ser solicitado futuramente. XP utiliza o desenvolvimento incremental e sua capacidade de integração é garantida pelo desenvolvimento guiado por testes, dessa forma deve-se desenvolver somente o que é solicitado, pois assim estaremos desenvolvendo somente aqueles 22% de funcionalidades que realmente importam.

2.5.6 Programação em par

A programação em par é uma pratica que só traz vantagens, apesar que pode parecer o contrario já que temos dois programadores fazendo o mesmo trabalho, trabalhando no mesmo código, na mesma maquina ao mesmo tempo.

Na programação em par, um programador assume o mouse e o teclado em quando o outro revisa o código digitado e pensa em um nível mais estratégico.

“Duas cabeças pensam melhor que uma”. Com a programação em par os programadores discutem possíveis soluções e optam pela mais simples o que colabora para a simplicidade geral do sistema. O código gerado por duplas de desenvolvedores geralmente tem melhor design, menor numero de inconsistências.

Outra vantagem da programação em par é o nivelamento técnico da equipe de desenvolvimento, o conhecimento de todos é ampliado e distribuído entre os membros da equipe.

Através dessa pratica economiza-se tempo, pois pequenos erros como de uma expressão booleana são mais facilmente encontrados, erros que às vezes podem levar dias para serem resolvidos.

2.5.7 Desenvolvimento guiado por testes

Esse assunto terá um capitulo a parte.

2.5.8 Projeto de melhoria

Melhorar constantemente o sistema ao longo do tempo, [...]não nos preocupamos em construir o software perfeito, nem o design perfeito, nem o processo perfeito, mas sim em aperfeiçoar esses e outros aspectos dos projetos continuamente[...] (IMPROVEIT (Brasil). **Melhoria.**.).[21]

2.5.9 Integração Continua

Equipes XP programam em par e de forma isolada e dessa forma um problema pode ser encontrado, duas duplas podem realizar alterações em um mesma parte do código causando alguma falha na integração. Esse problema é enfrentado com diversas integrações durante o dia, os programadores integram o trabalho realizado em um pequeno período de tempo para manter o sistema e a equipe atualizada, dessa forma se existir alguma necessidade de correção, ela será de apenas uma ou duas horas de trabalho que foi realizado. (Beck, 2000).

[21] Disponível em: <http://improveit.com.br/xp/principios/melhoria>. Acesso em: 02 jun. 2012

2.5.10 Propriedade de código coletiva

Todo o código é de propriedade de toda a equipe e qualquer membro da equipe pode altera-lo sem a necessidade de pedir alteração, o que no primeiro momento pode parecer uma pratica perigosa e realmente seria, mas a segurança dessa pratica é garantida pelos testes unitários.

Se um membro da equipe encontrar uma oportunidade para refatorar o código ele deve fazê-lo.

> Em um projeto XP, os pares se revezam, as pessoas se revezam na formação dos pares e todos têm acesso e autorização para editar qualquer parte do código da aplicação, a qualquer momento. Ou seja, a propriedade do código é coletiva e todos são igualmente responsáveis por todas as partes. (IMPROVEIT (Brasil). **Código Coletivo)**[22]**.**

2.5.11 Padrão de codificação

Metáforas são usadas na tentativa de manter a integridade conceitual do sistema.

> "XP procura explorar ao máximo a utilização de metáforas, para que clientes e desenvolvedores sejam capazes de estabelecer uma vocabulário apropriado para o projeto, repleto de nomes representando elementos físicos com os quais os clientes estejam habituados em seu dia-a-dia, de modo a elevar a compreensão mútua."
>
> (PIRES, Cassio. **Extreme Programming: um novo conceito em Metodologia de Desenvolvimento.)**[23]

Existe um ganho real na compreensão do sistema através das metáforas, por isso XP passou a adota-las. E bem comum enfrentarmos dificuldades em explicar um assunto à outra pessoa, principalmente ser for de uma área diferente, mas quando comparamos o assunto com algo de conhecimento geral, tudo fica mais simples.

2.5.12 Ritmo sustentável

XP propõe adotar um ritmo sustentável para o desenvolvimento, deixando de lado crenças que horas extras aumentaram a produtividade. "Seres humanos não se comportam como máquinas, portanto se cansam e produzem resultados indesejáveis em função da fadiga."

(TELES, 2005, p. 122)

---

[22] Disponível em: <http://improveit.com.br/xp/praticas/codigo_coletivo>. Acesso em: 02 jun. 2012.

[23] Disponível em: <http://www.gojava.org/files/artigos/ExtremePrograming.pdf>. Acesso em: 02 jun. 2012.

Consertar esses resultados indesejáveis decorrentes do cansaço acaba gerando atrasos ao invés de antecipar a entrega.

3 TESTE DE SOFTWARE

Testar o software é forma que temos para encontrar erros (Pressman, 2006, pg316), quando testamos um software buscamos encontrar diferenças entre os resultados obtidos e os resultados esperados (GUERRA, Ana Cervigni; COLOMBO, Regina Maria Thienne. **Tecnologia da Informação: qualidade de produto de software.)** [24]·, verificamos se ele atingiu suas especificações, se preenche os requisitos ao qual foi projetado.

Esse capítulo apresenta os principais tipos de teste de software e os métodos para testar software.

3.1 Métodos de Teste

Para se testar software temos dois métodos conhecidos como Caixa-branca e Caixa-preta e esses métodos não são mutuamente exclusivos (Santos 2010).

3.1.1 Teste Caixa-preta

Teste de Caixa-Preta as valida entradas e saídas sem se preocupar como o sistema chegou a esses resultados, ou seja, a estrutura pouco importa. A grande vantagem dessa abordagem esta em se focar em resultados, se o sistema entrega as saídas que dele se espera, por outro lado isso se torna um problema, pois desconsidera completamente a implementação do software, deixa de avaliar se as escolhas para chegar até o resultado foram as mais eficientes e condizentes com as boas praticas de programação.

"Teste de caixa-preta refere-se a teses que são conduzidos na interface do software. Um teste de caixa-preta examina um aspecto fundamental do sistema, pouco se preocupando com a estrutura lógica interna do software." (PRESSMAN, 2006, p.318).

3.1.2 Testes Caixa-branca

Testes de caixa branca são testes criados para testar o código de um software, esses testes buscam avaliar os possíveis caminhos lógicos que podem ser executados pelo sistema.

O Teste Unitário é um bom exemplo de teste de caixa branca, pois o teste unitário testa uma parte específica conhecida do código, validando as entradas e saídas geradas por aquele pedaço de código.

[24] Disponível em: < **http://hdl.handle.net/10691/151**>. Acesso em: 19 jun. 2012.).

“Teste caixa-branca de software é baseado em um exame rigoroso do detalhe procedimental. Caminhos lógicos internos ao software e colaborações entre componentes são testados, definindo-se casos de teste que exercitam conjunto de condições e/ou ciclos.”(PRESSMAN, 2006, p. 318).

A grande dificuldade na utilização dessa forma de testes é a inviabilidade de testar todos os caminhos lógicos possíveis em sistemas complexos, algumas centenas de linhas de códigos podem gerar um numero de caminhos que uma vida humana não seria o suficiente para testa-lo.”(PRESSMAN, 2006, p. 318).

3.2 Principais Tipos de Teste de Software

3.2.1 Testes Unitários

Testes de unidade ou testes unitários avaliam a menor parte de um sistema, ou seja, uma parte do código, um componente ou uma classe. Esses testes podem ser escritos pelo analista de testes ou pelo próprio desenvolvedor e examinam principalmente as entradas e saídas de dados. Esse tipo de teste é o teste utilizado na pratica TDD.

“Kent Beck introduziu o conceito de teste de unidade em Smalltalk, e tem levado a cabo em muitas outras linguagens de programação, tornando testes unitarios uma prática extremamente útil em programação de software” (OSHEROVE, 2009, p. 4).

3.2.2 Testes de Aceitação

Testes de aceitação são testes realizados no sistema antes da implantação onde o comportamento do sistema é examinado, onde o cliente testa e avalia se as entradas e saídas são as esperadas como especificadas nos requisitos. Testes de aceitação também são chamados de testes Alfa e Beta por poderem ser executados em um ambiente de testes que simula o ambiente real ou no próprio ambiente de real utilização.

3.2.3 Testes de Integração

Testes de integração são testes construídos para avaliar a interação entre componentes, esses testes são fundamentais quando se desenvolve incrementalmente ou sistema, pois são necessárias varias integrações, se for um projeto utilizando uma metodologia ágil como XP pode-se realizar varias integrações durante um mesmo dia.

Um dos benefícios do TDD segundo seus praticantes é o baixo acoplamento entre os componentes desenvolvidos isso gera uma grande na integração, com componentes mais independentes a possibilidade de incompatibilidade entre os componentes diminui.

3.2.4 Testes de Regressão

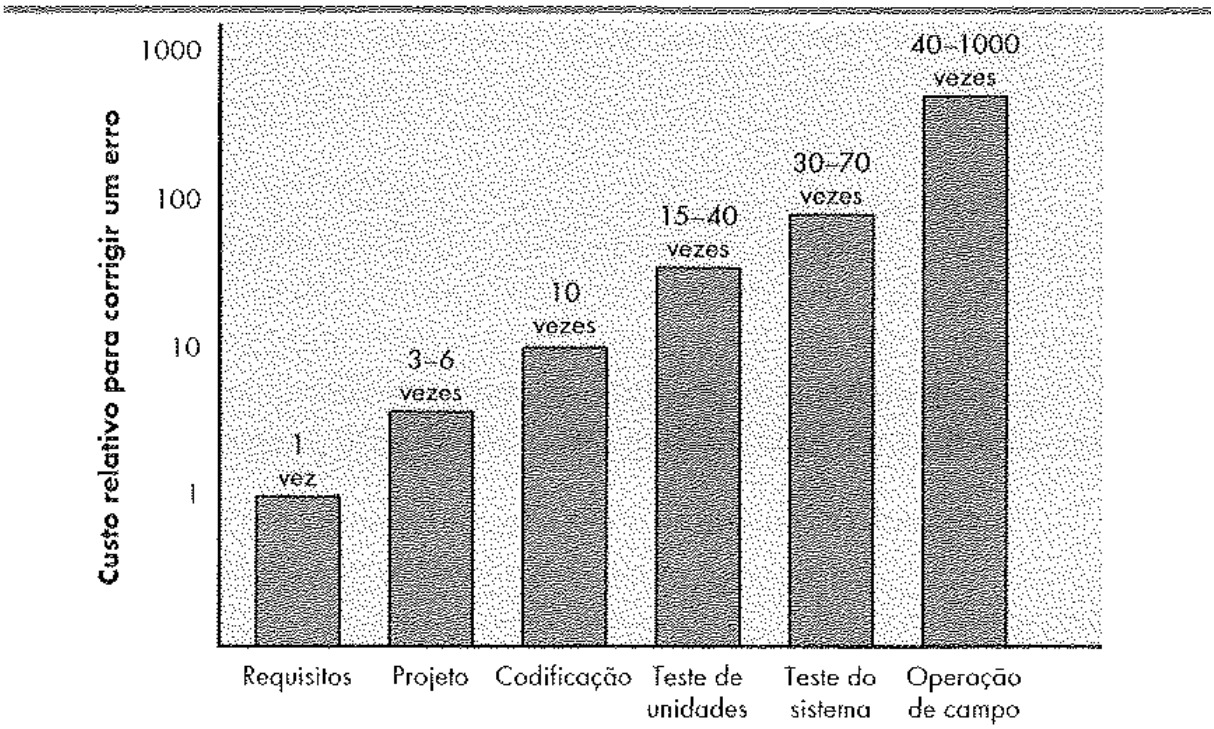
Após a integração de novos componentes ao sistema faz-se necessário testar novamente as outras partes do sistema para examinar a possibilidade de efeitos colaterais indesejáveis.

> Cada vez que um novo modulo é adicionado como parte do teste de integração, o software se modifica. Novos caminhos de fluxo de dados são estabelecidos, novas E/S pode ocorrer e nova lógica de controle é adicionada. Essas modificações podem causar problemas com funções que previamente funcionavam impecavelmente. (PRESSMAN, 2006, p. 300).

4 DESENVOLVIMENTO GUIADO POR TESTES

Falhas de software são responsáveis por grandes perdas financeiras todos os anos e segundo o relatório (DEPARTMENT OF COMMERCE (Usa). **Software Errors Cost U.S. Economy $59.5 Billion Annually.**) [25]esses custos poderiam ser reduzidos com uma infraestrutura melhor para identificar erros mais cedo, pois os custos de correção de erros aumentam exponencialmente com o desenvolvimento do sistema.

Figura 4 - Custos dos erros



Fonte: Pressman, 2006, pg 580.

Segundo Borges, (N. Borges, Eduardo. **Conceitos e benefícios do Test Driven Development**.) [26] o TDD foi criado visando à identificação mais cedo dos erros e consequentemente a diminuição dos custos e o aumento da qualidade do software desenvolvido.

[25] Disponível em: <http://web.archive.org/web/20090610052743/http://www.nist.gov/public_affairs/releases/n02-10.htm>. Acesso em: 02 jun. 2012.

[26] Disponível em http://www.inf.ufrgs.br/~cesantin/TDD-Eduardo.pdf. Acesso em: 22/09/2011.

4.1 Teoria

Apesar do nome desenvolvimento guiado por testes, apesar do TDD utilizar testes e auxiliar na identificação de erros, TDD não pode ser visto como uma forma de testar software (Santin, Carlos Eduardo**. Desenvolvimento guiado por testes e ferramentas xUnit**.), [27]não pode ser visto como o objetivo do TDD.

Segundo Ron Jeffries (JEFFRIES, Let Ron. **What is Extreme Programming?**) [28]o objetivo do TDD é conseguir código limpo que funcione e isso é alcançado escrevendo os testes antes do código, o que não chega a ser uma ideia nova, pois há muito tempo programadores simulavam entradas e saída de dados antes de começar a programar efetivamente. TDD usa essa ideia antiga junto com um conjunto de novas linguagens e ferramentas de programação visando obter um código funcional e com qualidade (PRANCHES, Henrique Feliciano. **Uma Avaliação Empírica de um Ambiente Favorável para o Desenvolvimento Dirigido por Testes.** 2007. 117 f. Tese (Mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2007.).

Existem estudos que evidenciam uma maior qualidade de softwares desenvolvidos uilizando a pratica TDD com números impressionantes, como um estudo de caso realizado pela IBM onde a adoção da pratica TDD teve um resultado de diminuição de 50% de defeitos encontrados em um projeto comparando com um sistema semelhante com um impacto mínimo sobre tempo e produtividade (WILLIAMS, Laurie; MAXIMILIEN, E. Michael. Assessing Test-Driven Development at IBM. In: ICSE 03, 25., 2003, Portland. **Artigo.** Washington: Ieee Computer Society, 2003. p. 564 - 569.).

4.2 Ciclo TDD

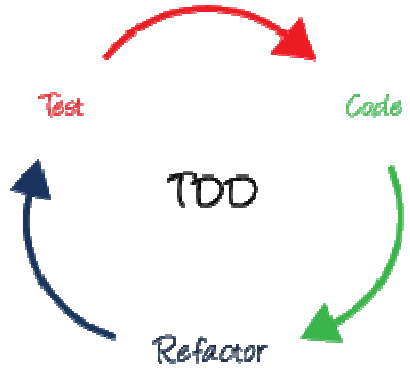
O desenvolvimento guiado por testes segue um ciclo de três regras básicas para sua implementação (Beck, 2010).

---

[27] Disponível em http://www.inf.ufrgs.br/~cesantin/TDD_Artigo.pdf. Acesso em: 22/09/2011.

[28] Disponível em: <http://xprogramming.com/what-is-extreme-programming/#planning>. Acesso em: 02 jun. 2012.

Figura 5 - Ciclo TDD



Fonte: http://rafaelfranchi.wordpress.com/2012/01/25/o-que-e-tdd/

1. Vermelho – Escrever um teste que falhe, esse teste não precisa nem compilar
2. Verde – Faça o teste funcionar o mais rápido possível, mesmo que isso implique em utilizar soluções não recomendáveis, como fixar constantes, o importante é o passar.
3. Refatorar – Esse é o momento de se preocupar com o design, remover todas as duplicações e simplificar o código.

> O TDD é um processo iterativo, portanto estes três passos anteriormente citados devem ser repetidos até que o desenvolvedor esteja satisfeito com o novo código gerado.A forma que o TDD trabalha é decompor os requerimentos do sistema em um conjunto de comportamentos necessários para cumprir estes requerimentos, sendo que para cada comportamento do sistema a primeira coisa a ser feita é escrever um teste de unidade para testá-lo. (Santin, Carlos Eduardo**.** **Desenvolvimento guiado por testes e ferramentas xUnit**.)[29]

4.3 Lista de Testes

Uma boa pratica para aplicar o TDD é iniciar construindo uma lista de testes, essa lista você ira modificando dinamicamente incrementando novos testes conforme as necessidades forem aparecendo.

4.4 Assertivas

Assertivas são as formas utilizadas para testar, passando entradas e saídas esperadas e o método para testar o resultados, frameworks com NUnit e JUnit possuem uma grande

[29] Disponível em http://www.inf.ufrgs.br/~cesantin/TDD_Artigo.pdf.
Acesso em: 22/09/2011.

quantidade de assertivas prontas para auxiliar o desenvolvedor no momento de escrever seu testes.

4.5 Objetos Simulados

O desenvolvimento guiado por testes prega que tudo pode ser testado entretanto testar certos itens pode não ser viável, um exemplo clássico é o banco de dados que também pode ser uma fonte de erros.

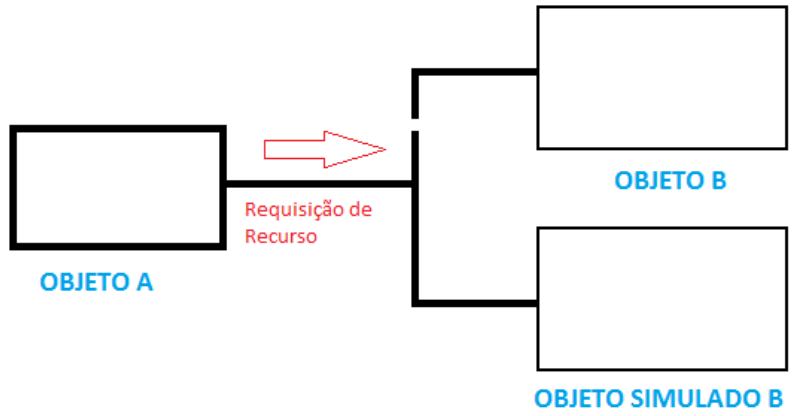
O banco de dados pode estar em produção ou em um servidor na rede o que pode dificultar ainda mais realizar testes e alterações.

TDD enfrenta esse cenário com Objeto Simulado (Mock Object em inglês), objetos que simulam o comportamento de um objeto real.

A utilização de objetos simulados evidentemente nos gera uma preocupação, será que os objetos simulados realmente ira se comportar como um objeto real?

“Você pode reduzir essa estratégia tendo um conjunto de testes para o Objeto Simulado que pode também ser aplicado ao objeto real quando tornar-se disponível”. (BECK, 2010, p. 165).

Figura 6 - Objeto Simulado



A imagem anterior mostra uma funcionalidade utilizando um Objeto Simulado, o objeto A faz a requisição de um recurso para o objeto B e um Objeto Simulado B deverá retornar um comportamento idêntico ao do objeto B para o objeto A.

4.6 Stubs

Da mesma forma que os Objetos Simulados, Stubs servem para simular objetos reais, mas Stubs utilizam outra abordagem.

> Com stubs, nos preocupamos em testar o **estado dos objetos** após a execução do método. Neste caso incluímos os asserts para ver se o método nos levou ao estado que esperamos.
>
> Com mocks, a preocupação é testar a **interação entre objetos** durante a execução do método. Neste caso, os asserts servem para ver se os métodos se relacionaram como o esperado. (SANCHEZ, Ivan. **Mocks vs Stubs: qual a diferença afinal?**)[30]

Trabalhar com Stubs, Objetos simulados e assertivas, seria impraticável sem fermentas que automatizassem esse processo e para enfrentar essa necessidade surgiram as ferramentas XUnit.

[30] Disponível em: <http://dojofloripa.wordpress.com/2006/10/20/mocks-vs-stubs-qual-a-diferenca-afinal/>. Acesso em: 02 jun. 2012

5 Frameworks XUnit

O desenvolvimento guiado por testes utiliza testes automatizados e essa automatização e feita através de frameworks conhecidos como XUnit.

Nesse capitulo será feita uma breve apresentação das principais ferramentas XUnit que são utilizadas para automatizar os testes unitários do TDD e será apresentado um exemplo de um código sendo desenvolvido com TDD e a ferramenta NUnit onde será possível visualizar passo a passo o ciclo TDD.

5.1 SUnit

O primeiro framework XUnit foi desenvolvido por Kent Beck para a linguagem Smaltalk e ficou conhecido pelo nome SUnit.

> A versão original do SUnit baseava-se na criação de pequenos testes, um para cada classe, para que caso algum erro ocorresse durante a implementação,este fosse rapidamente detectado e corrigido. E devido a essa característica, de os testes serem implementados por classe, a alteração em uma determinada classe resulta na alteração do teste relacionado apenas aquela classe e não dos demais. (SANTIN, Carlos Eduardo. **Desenvolvimento guiado por testes e ferramentas xUnit)**[31]

Atualmente existem frameworks para testes unitários automatizados como DUnit para Delphi Borland, CppUnit para C++ e os mais famosos NUnit para linguagens Dot Net da Microsoft e JUnit para linguagem Java,a maioria das linguagens devido a popularização dos testes unitários com TDD conta com seus próprios frameworks para automatizar testes.

A grande vantagem em automatizar testes é conseguir rodar todos os testes repetidamente, sem a interferência humana, dessa forma sem nenhum esforço verificamos as funcionalidades do sistema rapidamente e temos um feedback imediato de efeitos colaterais.(BERNARDO, Paulo Cheque; KON, Fabio. **A Importância dos Testes Automatizados**.)[32]

O próximo tópico demonstra a execução do ciclo TDD em conjunto com uma framework de testes automatizados.

[31] **.** Disponível em: <http://www.inf.ufrgs.br/~cesantin/TDD_Artigo.pdf>. Acesso em: 02 jun. 2012.
[32] **.** Disponível em: <http://www.ime.usp.br/~kon/papers/EngSoftMagazine-IntroducaoTestes.pdf>. Acesso em: 24 jun. 2012

5.3 Exemplo do Cliclo TDD com NUnit

O exemplo abaixo demonstra um simples código para verificar se um numero é primo

Figura 7 - Codigo Primo 1

```
using System;
using NUnit.Framework;
using Numeros_Primos;


namespace Test_Numeros_primos
{
/*Lista de Testes
 * 1 - Verificar se o numero é primo
 */
    [TestFixture]
    public class TestPrimos
    {
        [Test]
        public void IsPrimo()
        {
            Avaliacao_de_numeros_primos np = new Avaliacao_de_numeros_primos();
            Assert.IsTrue(np.Avaliar_Primo(3));
        }

    }
}
```

A imagem acima mostra a criação do nosso primeiro teste, podemos notar que o compilador do SharpDeveloper acusa um erro na chamada do método Avaliar_Primo() pois o mesmo não existe ainda. Na abordagem TDD isso não é um problema, estamos no caminho certo porque o primeiro passo é criar um teste que falhe, esse teste não necessariamente precisa compilar.

Após criarmos um teste que falhe, devemos fazer com que ele funcione o mais rápido possível.

Figura 8 - Codigo Primo 2

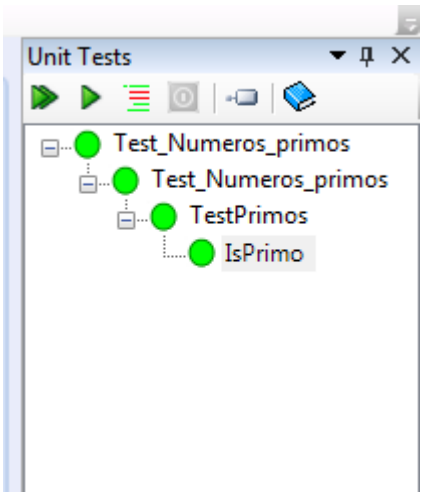
```
using System;

namespace Numeros_Primos
{
    /// <summary>
    /// Description of Avaliacao_de_numeros_primos.
    /// </summary>
    public class Avaliacao_de_numeros_primos
    {
        public Avaliacao_de_numeros_primos()
        {
        }

        public bool Avaliar_Primo(int numero)
        {
            return true;
        }
    }
}
```

A imagem acima mostra a criação do método para fazer o teste funcionar da forma mais simples possível. Vemos então nossa primeira barra verde fornecida pelo nosso framework de testes.

Figura 9 - Codigo Primo 3



Apesar de o teste funcionar, ele ainda não esta validando a regra dos números primos, pois qualquer valor passado como parâmetro, sempre retornara com true, ou seja, qualquer numero será identificado como primo.

Ao chegarmos a esse ponto entramos em uma outra etapa, a fase de refatoração, começamos então a remover duplicações, substituir constantes por variáveis e aplicar as regras de negocio.

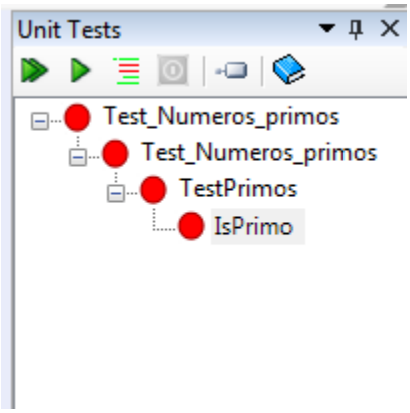
Como sabemos, números primos são números naturais, ou seja, não possuem valores negativos, começamos aplicando essa regra.

Figura 10 - Código Primo 4

```
public bool Avaliar_Primo(int numero)
{
    if (numero < 0) {
        return false;
    }
    return true;
}
```

Realizando a chamada do método com parâmetro negativo, Avaliar_Primo(-3), nossos testes não passam, comportamento que esperamos deles.

Figura 11 - Código Primo 5



Nesse momento nosso código esta melhor, mas ainda não valida nossos regras de negocio.

Números primos são aqueles que são divisíveis somente por 1 e pelo próprio numero, tratamos os dois casos de números primos triviais, 1 e 2.

Figura 12 - Código Primo 6

```
public bool Avaliar_Primo(int numero)
{
    if (numero < 0) {
        return false;
    }
    if ((numero > 0) && (numero < 3))
    {
        return true;
    }
    return true;
}
```

Resta agora tratarmos os casos para números maiores que dois.

Figura 13 - Codigo Primo 7

```
public bool Avaliar_Primo(int numero)
{
    bool isprimo = true;

    if (numero < 0) {
        return false;
    }
    if ((numero > 0) && (numero < 3))
    {
        return true;
    }
    for (int i = 2; i < numero; i++)
    {
        if (numero % i == 0)
        {
            isprimo = false;
        }
    }
    return  isprimo;
}
```

Conseguimos um código que valida todas as regras de números primos, podemos então refatorar o código com a segurança de que qualquer alteração que comprometa a validação das regras de negocio serão alertadas pelos testes que foram criados.

Revisando o código, o IF para validar os números 1 e 2 aparentemente não se fazem necessários, o laço FOR que criamos, não é executado para valores 1 e 2 e o método retorna o valor padrão da variável isprimo, que esta iniciada com true.

Figura 14 - Codigo Primo 8

```
public bool Avaliar_Primo(int numero)
{
    bool isprimo = true;

    if (numero < 0)
    {
        return false;
    }

    for (int i = 2; i < numero; i++)
    {
        if (numero % i == 0)
        {
            isprimo = false;
        }
    }
    return  isprimo;
}
```

O código acima é o código refatorado, o código final que valida nossas regras e continua passando em todos os testes.

6 Estudo de Caso

Este capítulo apresenta um comparativo entre dois projetos desenvolvidos, um utilizando a pratica TDD e outro desenvolvido de forma convencional sem TDD.

Serão apresentados valores de métricas comparando os resultados das duas abordagens para evidenciar se existiu alteração na qualidade do software desenvolvido com TDD.

6.1 Hospital S*

Na década de 1960, um grupo de nove pediatras idealizou a criação de um Pronto Socorro infantil, objetivo que foi concretizado em 14 de abril de 1962 com a inauguração do Pronto Socorro em um casarão no bairro de Higienópolis.

Em 1972 o hospital S passou a funcionar em sede própria, passou a atendem em cinco pavimentos localizados a menos de cem metros de seu antigo endereço.

Em 2005 parte do hospital S foi comprado pelo Dr. Jose Luiz Setubal atual presidente e herdeiro do grupo Itaú o qual foi responsável por sua reestruturação que transformou o hospital S deu m Pronto Socorro modelo em um hospital de excelência reconhecida no atendimento infantil.

Hoje o hospital S é um dos mais renomados hospitais da cidade, funcionando em sua cede própria, um edifício de dezessete andares.

6.2 Empresa F*

Empresa com mais de 10 anos de experiência no desenvolvimento de soluções personalizadas para atendimento presencial, virtual e telefônico, atuando em todo o território nacional.

A empresa F desenvolve sistemas para todos os seguimentos, mas em especial para o setor público, hospitais, universidades e empresas de telecomunicações.

6.3 O projeto sem papel[33]

Alguns hospitais tem adotado a pratica “Hospital sem papel” que consiste em abolir o uso do papel no atendimento medico, entretanto essa pratica geralmente resume-se a substituir prontuários de papel por sistemas on line.

* Utilizados nomes fictícios devido à ausência de autorização das empresas.

O hospital S possui um projeto para abolir completamente o uso do papel e para isso ira investir mais de um milhão de reais na aquisição de software, treinamento, revisão dos processos entre outras atividades visando atingir esse objetivo.

A empresa F foi uma das empresas contratada para implantar sua ferramenta de gerenciamento de atendimento e filas e foi solicitado um protótipo de sistema de pesquisa de opinião, para substituir a forma atual de preenchimento de formulário de papel.

6.4 Ambiente de utilização do sistema

O sistema de pesquisa de opinião devera funcionar pelo período de trinta dias no hall do hospital, nesse local não existe pontos de rede próximos e a área de TI não disponibilizou internet Wi-Fi, dessa forma o sistema funcionara localmente sem acesso a rede ou internet.

Foram desenvolvidas duas soluções para persistir os dados, uma utilizando um banco de dados e outra guardando os arquivos em um arquivo XML.

Nesse estudo de caso será utilizada para comparação a solução com arquivo XML.

6.5 Ambiente de Desenvolvimento

Métricas de software é assunto polemico dentro da área de TI, pois geralmente seus números não são absolutos e comparar sistemas, mesmo que semelhantes se torna complicado.

Para avaliar a pratica TDD, será comparado será comparado o mesmo sistema, o modulo de pesquisa de opinião.

O sistema de pesquisa de opinião foi desenvolvido pela empresa F sem a utilização do TDD por um desenvolvedor Jr e agora quatro meses depois o mesmo sistema será refeito pelo mesmo desenvolvedor, agora um iniciante na pratica TDD.

6.6 Métricas de Software

6.6.1 Indice de Manutenabilidade

Indice que varia de 0 a 100, sendo quando maior o valor, melhor a possibilidade de manutenção do código

- 0 – 9 Baixa manutenção ou critica
- 10 – 19 Manutenção de nível moderado
- 20 – 100 Bom nível de manutenção

Formula:

= MAX(0,(171 - 5.2 * log(Halstead Volume) - 0.23 * (Complexidade Ciclomatica) - 16.2 * log(Linhas de código))*100 / 171)(ZAIN NABOULSI. Code Metrics – Maintainability Index)[34]

6.6.2 Complexidade Ciclomática

Mede a complexidade da estrutura do sistema, verifica o numero de caminhos independentes no código.

Formula:

M = E – N + 2*P

M = Complexidade Ciclomática

E = Quantidade de setas

N = Quantidade de nós

P = Quantidade de componentes conectados

(FERRAZ, Ronaldo Melo. **Conceitos de Programação: Complexidade Ciclomática.)**[35]

6.6.3 Acoplamento de classes

Numero de referencias a outros objetos

Formula:

Soma de chamadas para outros objetos

6.6.4 Cobertura de testes

Linhas de código testadas

Formula:

Total de linhas testadas em porcentagem

[34] . Disponível em: <http://blogs.msdn.com/b/zainnab/archive/2011/05/26/code-metrics-maintainability-index.aspx>. Acesso em: 09 jun. 2012.

[35] Disponível em: <http://www.sigmaaldrich.com/catalog/product/sigma/30408?lang=pt®ion=BR>. Acesso em: 18 maio 2012.

6.7 Valores coletados

Todos os dados foram obtidos através da ferramenta de testes Code Metrics do Microsoft Visual Studio 2010 com suas configurações default e tabelados abaixo.

6.7.1 Índice de Manutenibilidade

A ISO/IEC 9126 – 1 define manutenibilidade como a Capacidade do produto de software de ser modificado. As modificações podem incluir correções, melhorias ou adaptações do software devido a mudanças no ambiente e nos seus requisitos ou especificações funcionais, segundo a ISO/IEC 9126 a manutenibilidade é um dos atributos da qualidade de software.

Índice de manutenibilidade			
	Mínimo	Media	Maximo
TDD	63	86,4	98
Sem TDD	60	87,375	98

Tabela 1 – Tabela de Índice de Manutenibilidade

6.7.2 Complexidade Ciclomática

A complexidade ciclomática (CC) mede a complexidade do código, quanto mais complexo um código, mais difícil de alterar, manter e alterar e testar o código se torna. (SANTOS FILHO, Firmino Dos. **MÉTRICAS E QUALIDADE DE SOFTWARE**)[36]

Complexidade Ciclomática			
	Mínimo	Media	Maximo
TDD	1	1,6	5
Sem TDD	1	1,125	2

Tabela 2 – Tabela de Complexidade Ciclomática

6.7.3 Acoplamento

O índice de acoplamento é um fator importante na qualidade de software porque um alto acoplamento entre os componentes pode gerar erros em efeito cascata e se os desenvolvedores não podem estimar o impacto de uma alteração, seu custo também não pode ser estimado. (MOREIRA, 2011).

Acoplamento			
	Mínimo	Media	Maximo
TDD	0	1,1	3
Sem TDD	0	0,875	6

Tabela 3 – Tabela de Acoplamento

[36] . Disponível em: <http://www.angelfire.com/nt2/softwarequality/Quality/>. Acesso em: 24 jun. 2012.

6.8 Analise dos Resultados

Uma das vantagens do TDD segundo seus praticantes é a simplicidade do código gerado e a facilidade de manutenção e alteração garantidas pelos testes unitários, entretanto não foram identificadas nesse estudo de caso diferenças significativas entre as abordagens com TDD e sem. Os valores dos índices de manutenibilidade foram bem semelhantes, apenas a pior nota nesse quesito da abordagem sem TDD foi pior do que a utilizada com a abordagem com TDD.

A complexidade ciclomática do código com TDD, foi na media superior ao projeto sem TDD e também nesse quesito foi encontrada a pior nota do TDD, o método mais complexo do projeto recebeu uma nota 5 enquanto sem TDD a pior nota foi 2. O mercado de TI tem usado o numero 10 como numero limite para esse teste, apesar de não existir nenhuma base cientifica para esse numero apenas e comumente aceito.

O TDD prega que os testes unitários levam ao menor acoplamento do código gerado, nesse teste a abordagem sem TDD teve sua pior nota comparada com o projeto utilizando TDD, o pior método do projeto sem TDD teve nota 6 chegando próximo a 9, valor considerado pelo mercado como limite para acoplamento,o projeto com TDD teve sua pior nota nesse quesito 3 apesar da media do projeto sem TDD ter obtido um valor ligeiramente menor.

Aos dois projetos foi aplicada uma lista de testes para verificação de requisitos e validação, nesse ponto a pratica TDD teve seu melhor desempenho. O projeto com TDD passou em 70% dos casos de teste frente e obtiveram 50% menos erros que o projeto sem TDD.

Os comparativos entre as duas abordagens evidenciam que nesse estudo de caso, o projeto utilizando TDD foi mais eficiente no tratamento dos erros, obtendo 50% menos erros assim como no estudo da IBM(WILLIAMS, Laurie; MAXIMILIEN, E. Michael. Assessing Test-Driven Development at IBM. In: ICSE 03, 25., 2003, Portland. **Artigo.** Washington: Ieee Computer Society, 2003. p. 564 - 569.).

, dessa forma o TDD apresenta-se com uma importante ferramenta para aumentar a qualidade do software e diminuir os custos de correção de erros.

7 Conclusão

A utilização do TDD garante uma qualidade melhor na identificação de erros mais cedo, o que por si só já é um motivo valido para sua adoção, pois melhora a qualidade do software e diminui custos e tempo para manutenção.

A pratica TDD teve resultados relevantes no produto final, contendo menos erros e menor acoplamento entre os componentes por isso conclui-se que é uma abordagem valida visando um aumento da qualidade dos sistemas desenvolvidos.

Referencias

AGILE MANIFEST. **Agile Manifest.** Disponível em: <http://agilemanifest.org>. Acesso em: 03 jun. 2012

BECK, Kent. **TDD Desenvolvimento Guiado por Testes.** Porto Alegre: Bookman, 2010.

BERNARDO, Paulo Cheque; KON, Fabio. **A Importância dos Testes Automatizados.** Disponível em: <http://www.ime.usp.br/~kon/papers/EngSoftMagazine-IntroducaoTestes.pdf>. Acesso em: 24 jun. 2012.

DEPARTMENT OF COMMERCE (Usa). **Software Errors Cost U.S. Economy $59.5 Billion Annually.** Disponível em: <http://web.archive.org/web/20090610052743/http://www.nist.gov/public_affairs/releases/n02-10.htm>. Acesso em: 02 jun. 2012.

DEVMEDIA (Brasil). **Engenharia de Software - Introdução a Teste de Software Leia mais em: Artigo Engenharia de Software - Introdução a Teste de Software http://www.devmedia.com.br/artigo-engenharia-de-software-introducao-a-teste-de-software/8035#ixzz1wexTAqmT.** Disponível em: <http://www.devmedia.com.br/artigo-engenharia-de-software-introducao-a-teste-de-software/8035>. Acesso em: 02 jun. 2012.

EXTREME PROGRAMING.ORG (Usa**). Make frequent small releases**. Disponível em: <http://www.extremeprogramming.org/rules/releaseoften.html>. Acesso em: 02 jun. 2012.

FERRAZ, Ronaldo Melo. **Conceitos de Programação: Complexidade Ciclomática.** Disponível em: <http://www.sigmaaldrich.com/catalog/product/sigma/30408?lang=pt®ion=BR>. Acesso em: 18 maio 2012

Gaspareto, Otávio. **Test Driven Development**.

Disponível em: http://www.inf.ufrgs.br/~cesantin/TDD-Otavio.pdf.Acesso em: 22/09/2011.

GUERRA, Ana Cervigni; COLOMBO, Regina Maria Thienne. **Tecnologia da Informação: qualidade de produto de software.** Disponível em: < **http://hdl.handle.net/10691/151**>. Acesso em: 19 jun. 2012.

Gil, Antonio Carlos. **Como Elaborar Projetos de Pesquisa**. São Paulo: Atlas, 1991.

IMPROVEIT (Brasil). **Melhoria.** Disponível em: <http://improveit.com.br/xp/>. Acesso em: 02 jun. 2012.

JEFFRIES, Let Ron. **What is Extreme Programming?** Disponível em: <http://xprogramming.com/what-is-extreme-programming/#planning>. Acesso em: 02 jun. 2012.

MARTIN, Robert. **Código Limpo.** Rio de Janeiro: Alta Books, 2011.

MOREIRA, Gabriel de Souza Pereira. **EARLY-FIX: UM FRAMEWORK PARA PREDIÇÃO DE MANUTENTENÇÃO CORRETIVA DE SOFTWARE UTILIZANDO MÉTRICAS DE PRODUTO.** 2011. 258 f. Tese (Mestrado) - Instituto Tecnológico de Aeronáutica, São José Dos Campos, 2011.).

N. Borges, Eduardo. **Conceitos e benefícios do Test Driven Development**.

Disponível em http://www.inf.ufrgs.br/~cesantin/TDD-Eduardo.pdf.

Acesso em: 22/09/2011.

OSHEROVE, Roy. **The art of unit testing.** Greenwitch: Manning Publications, 2009.

PRANCHES, Henrique Feliciano. **Uma Avaliação Empírica de um Ambiente Favorável para o Desenvolvimento Dirigido por Testes.** 2007. 117 f. Tese (Mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2007.

PIRES, Cassio. **Extreme Programming: um novo conceito em Metodologia de Desenvolvimento.** Disponível em: <http://www.gojava.org/files/artigos/ExtremePrograming.pdf>. Acesso em: 02 jun. 2012.

PRESSMAN, Roger S.. **Engenharia de Software.** 6. ed. Rio de Janeiro: Mcgraw-hill, 2006.

SANCHEZ, Ivan. **Mocks vs Stubs: qual a diferença afinal?** Disponível em: <http://dojofloripa.wordpress.com/2006/10/20/mocks-vs-stubs-qual-a-diferenca-afinal/>. Acesso em: 02 jun. 2012.

Santin, Carlos Eduardo**. Desenvolvimento guiado por testes e ferramentas xUnit**.

Disponível em http://www.inf.ufrgs.br/~cesantin/TDD_Artigo.pdf.

Acesso em: 22/09/2011.

SANTOS, Rafael Liu. **Emprego de Test Driven Development no desenvolvimento de aplicações.** 2010. 108 f. Dissertação (Bacharelado) - Universidade de Brasilia, Brasília, 2010.

Standish Group Chaos Report.

http://www.projectsmart.co.uk/docs/chaos-report.pdf

Acesso em: 22/09/2011

SOARES, Michel Dos Santos. **Metodologias Ágeis Extreme Programming e Scrum para o Desenvolvimento de Software.** Disponível em: <revistas.facecla.com.br/index.php/reinfo/article/download/146/38>. Acesso em: 24 jun. 2012.

TELES, Vinicius. **Um estudo de caso da adoção da praticas e valores do extreme programming.** 2005. 179 f. Tese (Mestrado) - Uf-rj, Rio de Janeiro, 2005.

WILLIAMS, Laurie; MAXIMILIEN, E. Michael. Assessing Test-Driven Development at IBM. In: ICSE 03, 25., 2003, Portland. **Artigo.** Washington: Ieee Computer Society, 2003. p. 564 - 569.